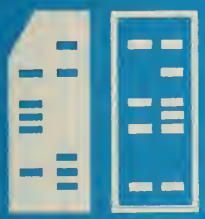
CONTROL STATEMENT SYNTAX AND SEMANTICS OF A LANGUAGE FOR PARALLEL PROCESSORS

by

Robert B. Wilhelmson

January 13, 1969



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. 298

CONTROL STATEMENT SYNTAX AND SEMANTICS OF A LANGUAGE FOR PARALLEL PROCESSORS*

by

Robert B. Wilhelmson

January 13, 1969

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

*This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. US AF 30(602)4144 and submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science, February, 1969.

ACKNOWLEDGMENT

The author would like to express his appreciation to his adviser, Dr. D. Kuck, for his suggestions and criticisms in the course of the study. The author is also indebted to his co-workers, Paul Budnik, Yoichi Muraoka, and Norma Abel, for their criticisms and suggestions.

Thanks are also extended to the many who have helped prepare the final manuscript.

ABSTRACT

A description of the control statement features of TRANQUIL, a language for describing algorithms in terms of parallel constructs, and their implementation in a compiler for the parallel array computer ILLIAC IV is given. This includes descriptions of the revised ALGOL loop specification, simultaneous execution specification, and set specification; and the implementation of the storage and use of sets as well as use of the control unit registers. The concept of parallelism in a language is discussed and the problems of automatic translation of a serial to a parallel language brought out.

TABLE OF CONTENTS

LIST OF FIGURES

1. PARALLELISM IN REVIEW

1.1 Parallel Concepts

Today the use of the term parallelism in the field of computer science can, unless clarified, result in confused thinking due to its possible use in referring to a variety of concepts. In the early years of computer development the term became associated with the simultaneous handling of input-output processes and processor computations. This simultaneous capability reduces processor idle time; however, it is often the case that it cannot be used because of the serial nature of input-output requests and computation. The introduction of the parallel concept known as multiprogramming was directed in part to overcoming this problem by allowing several programs to share a processor during their execution stage. The implementation of this concept requires efficient and balanced handling of input-output processes, memory allocation, scheduling, and queueing. Without careful planning the time gained can easily be offset by the time lost. An aid in this regard is the use of re-entrant code; i.e., code that can be used on demand, and not when some other user is finished with it.

From the hardware perspective emerges yet another parallel concept, the utilization of a fixed number of processors in what is called a multiprocessing system. The implementation of this concept has been most successful when the processors are associated with certain tasks (thus lifting the burden off a single unit) or when

programs are allocated to processors on a one-to-one basis. However, their use for executing sections of a program that are data independent over a series of operations has not been very successful. To accomplish this requires the ability to specify in or decide from a language on the level of FORTRAN or ALGOL this independence [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]. In the former case the problem centers around attempting to denote parallel operations in serial form and in the latter the task of algorithmically rethinking in parallel a program that is specified serially is unsolved.

Another parallel concept concerning hardware is the use of a large number of processing elements under a single control unit. Each processing element responds to a given operation simultaneously with the others unless it is disabled. A system using this concept is called an array processing system and the ILLIAC IV computer falls into this category. This type of system can be designed using one large memory or individual processing element memories with the problem of scheduling memory use or of moving information between separate memories occurring, respectively.

1.2 Explicit Parallelism

In writing a program it is at times clear to the programmer that several statements or sections of his code are independent of one another and thus can be executed simultaneously. The concepts of FORK and JOIN were introduced to allow the programmer to specify when this occurs.

Opler [3] presented these concepts for use in a multiprocessing environment by a single program in terms of the instructions DO TOGETHER and HOLD. These instructions are coupled with appropriate labels to specify the parallel paths and are to be used with the following rules:

(1) variables common to several parallel paths cannot be altered from within these paths,

(2) procedure calls from parallel paths in the same range must be made only to re-enterable code,

(3) branching into or out of the range of a DO TOGETHER or in between parallel paths is forbidden,

(4) logical decisions in parallel paths should set switches for later interrogation, and

(5) DO TOGETHER's may be nested.

Several months after the appearance of Opler's paper Anderson [4] suggested several extensions to ALGOL in a similar manner and noted that the FORK and JOIN notation could be used not only in multiprocessing systems but also in multiprogramming systems to designate independent segments of the same program so that while one waits for an I/O transaction another can be in execution. He introduced five delimiters in his ALGOL extension, namely: FORK, JOIN, TERMINATE, OBTAIN, and RELEASE.

The special words FORK and JOIN are followed by labels denoting the independent sections. When the programmer wants to

join independent sections, and not necessarily proceed on to the immediately succeeding statement, TERMINATE is used. This allows the FORK statement to be used recursively. OBTAIN allows the specification of variables for exclusive use in a particular path while RELEASE frees them. A simple example presented in Anderson's paper illustrating the use of FORK and JOIN is that of an inner product calculation broken up into two sections for vectors A and B.

```
        FORK  FIRST, LAST;
        FIRST:BEGIN  SI:=0;
              FOR I:=1 STEP 1 UNTIL N ÷ 2 DO
              SI:=SI + A[I] × B[I];
              GO TO CONTINUE;
              END;
        LAST: BEGIN  S2:=0;
              FOR J:=(N ÷ 2) + 1 STEP 1 UNTIL N DO
              S2:=S2 + A[J] × B[J];
              GO TO CONTINUE;
              END;
CONTINUE:JOIN  FIRST, LAST;
              S:=SI + S2
```

The advisability of using these terms and this means of expressing independent segments of a code has come under question. Wirth [5], in response to Anderson's article, distinguishes two cases of parallelism:

1) the use of processors in parallel and

2) The use of a processor configuration by a program requiring the use of various special components in parallel.

He protests the use of FORK and JOIN in ALGOL as contradictory to the principles of machine independence on which ALGOL is based since the JOIN statement is not logically necessary but is present for machine implementation. He introduces the use of and to denote the possibility of parallel execution in a program for case 1) with essentially the same effect as Anderson's FORK and JOIN. The problem in case 2) he states is the communication between different facilities requiring the access of common variables. This could be solved in one way by a special procedure including a number of programs, the first queueing requests to the others and permitting use of another program in the procedure only when all previous requests had been honored.

The topic of a recent paper by Constantine [6] is the control of sequence and parallelism in modular programs. He first presents a number of technical and programming considerations for the operation and specification of parallel processes and references appropriate papers. The technical considerations include:

(1) the prevention of simultaneous critical section usage in a shared code or resource,

(2) the assurance that no program will be delayed indefinitely, and

(3) the reproducibility of results under different conditions such as program mix and time.

The programming considerations include:

(1) simplicity of programmer specification, and

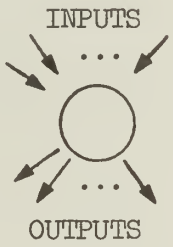
(2) allowing errors that are discoverable and correctable in a reasonable time.

His proposal for parallel activity in terms of modular (i.e. procedure level) units is a result of these considerations, its applicability to real problems, and the low overhead for implementation in comparison to that of the lower level FORK and JOIN. In particular, this proposal is that procedure input and output parameters should be isolated with the intent that the procedure call and its input specification be made as early as possible and the call for output parameters as late as possible.

1.3 Implicit Parallelism

The detection of parallelism implicitly from a code is desirable if possible in order that this parallelism might be utilized in multiprogramming and multiprocessing systems without having the user explicitly find and indicate it. This detection can be studied from several viewpoints. One approach is the development of a model for the description and analysis of parallel computations. This model has taken in several instances the form of a graph. In a 1964 paper by Karp and Miller [7] such a model is presented for the description and analysis of parallel computations. They include

several theorems on determinacy, termination, and queueing, where computation steps correspond to nodes of a graph and dependency between computation steps is represented by branches and associated queues of data. In another paper by Martin and Estrin [8] the nature of cycles in computed programs is discussed for parallel processors. Their model nodes again represent computational steps or processes and the areas connecting the nodes represent data flow associated with control conditions, branching probabilities, and data transfer time costs. The relation of sequential execution in these graphs is not separated from simultaneous execution. The basic unit is a node:



which represents a subprocess operating on inputs and yielding outputs. The actuation of a node via a single input or all inputs directed to the node can be specified as well as the output on one or all of the output lines. Significant results from studying implicit parallelism through this or any type of graph as yet do not exist.

A group at the Burrough's Corporation is studying the implicit detection problem from the standpoint of computer program

and multiprocessor relations [9, 10, 11]. They have presented algorithms to automatically detect the essential order between assignment statements involving variables, but it appears that the time involved for real problems is large even if hardware is used in implementation. They have also investigated the automatic recognition of possible parallel executions for a loop body and have described a system design to exploit both instruction and data parallelism with as yet no significant detailed practical results.

A related problem is the translation of a serial language into a parallel language for use on a parallel machine. An initial approach to the transformation of FORTRAN into TRANQUIL, the language for use on ILLIAC IV, appears in Appendix C. Before reading it, however, one should read the remaining chapters in order to become familiar with TRANQUIL, as a basic understanding of it is assumed there.

2. TRANQUIL CONTROL STATEMENTS

2.1 An Introduction to TRANQUIL and ILLIAC IV

TRANQUIL is the name of the algorithmic language which is being used to write programs for the array processing computer, ILLIAC IV. The specification of TRANQUIL is the first project to define a language for a real parallel machine and its development has proceeded side by side with hardware development. A general description of ILLIAC IV is given at the beginning of a paper by David J. Kuck [12] and is reproduced in the following paragraphs for later reference.

> ILLIAC IV is an array of 256 coupled computers driven by instructions from a common control unit. Each of the 256 processing elements (PEs) has 2048 words of 64-bit memory with a 240-ns cycle time. Each PE is capable of 64-bit floating-point multiplication in 400 ns and addition in 240 ns. 32-bit floating-point operations and 8-bit fixed-point operations are also available. The PE instruction set is similar to that of conventional machines, with two exceptions. First, the PEs are capable of communicating data to four neighboring PEs by means of routing instructions. Second, the PEs are able to set their own mode registers to effectively disable or enable themselves.
>
> ILLIAC IV has four control units (CUs) each of which may drive 64 PEs independently. The machine may thus be operated as four independent 64-PE quadrants or as two 128-PE halves. In the united configuration, all 256 PEs are effectively driven by one CU and routing proceeds across quadrant boundaries. This allows some flexibility in fitting problems to the array.
>
> The complete ILLIAC IV system includes a Burroughs B6500 computer which contains most of the software for the entire system. Also, there is a 10^9-bit, head-per-track disk with a 40-ms rotation speed and

an effective transfer rate of 10^9 bits per second. This allows the loading of the 32×10^6-bit ILLIAC IV memory from the disk in a transmission time of 32 ms. The input/output controller (IOC) contains a queuer for disk requests and a 2^{17}-bit buffer memory to smooth transmissions to and from the slower B6500 memory and other input/output devices.

In developing a language such as TRANQUIL for a parallel computer several considerations are necessary. The first is that it is desirable to be able to express algorithmic parallelism from the perspective of a problem and not the machine on which it is run. This does not mean that data structure mapping should not be a user concern but that once data is delegated to memory the user should only have to think in terms of his problem structures. The second is that the specification of parallelism should not be intricate, but straightforward and easy to learn and use so that all potential parallel activities can be specified. This is necessary if a parallel computer is to be utilized efficiently.

These considerations were made in the development of TRANQUIL. In particular, TRANQUIL is designed for algebraic problems and consists of a collection of data declarations, arithmetic and boolean operations, set definitions and operations, and control statements. The control statements are used to express parallel operations on chosen subsets of arrays or other geometric configurations. This is accomplished by linking control variables simultaneously with all the elements of associated sets. When the elements of a set are used simultaneously the actual implementation may take the form of PE indices or mode words. These sets can also be used in the specification of sequential or iterative control in much the same

way as in ALGOL. In contrast many loops can be eliminated from ALGOL code by using in TRANQUIL simultaneous control statements as mentioned and also by specifying operations on whole arrays such as the vector inner product or matrix multiplication operation. Further, there are control statements that allow branching on conditional boolean tests involving whole or partial arrays.

This then is a brief description of TRANQUIL control statements. The following section expands upon this description in manual form.

2.2 TRANQUIL Control Statement Manual

2.2.1 Introduction

The following manual is designed to help the TRANQUIL user become familiar with writing control statements in the language. A basic understanding of ALGOL is assumed and the approach is informal. The formal specification of the syntax is located in Appendix B and should be referred to when necessary.

Control statements are used in TRANQUIL to designate conditional and unconditional jumps, sequential loops, and simultaneous statement execution. Sets, which are an integral part of these control statements, are the subject of the next section.

2.2.2 Set Definitions, Declarations, and Operations

2.2.2.1 Defining Sets

Sets in TRANQUIL are collections of elements and are distinguished from vectors and arrays by their associated control functions.

A set is composed of ordered elements having integer values. These elements are n-tuples of arithmetic expressions ($0 \leq n \leq 7$) rounded via the entier function where n is constant for any given set and is known as the dimension of the set. For example,

SET1 ← [ ]

SET2 ← [1, 2, 3, 4, 5, 6, 7, 8]

SET3 ← [[10, 10], [9, 8], [8, 6], [7, 4], [6, 2]]

SET4 ← [25, -2, P, Q-R, 25]

where SET1 is empty and thus has dimension zero, SET2 and SET4 are 1-dimensional sets, and SET3 is 2-dimensional.

Sets can only be defined when not in use by a control statement and their definitions fall into two basic categories, list sets and comparison sets.

2.2.2.1.1 List Sets

The basic list set is specified by the explicit statement of its elements as in the examples just given. The specification of SET2 can be shortened to the increment form; i.e.,

SET2 ← [1, 2,..., 8]

The increment form can be used for any 1-dimensional set that is composed of elements either in a strictly increasing or strictly decreasing arithmetic progression. For example,

SET5 ← [6, 4, 2, 0, -2, -4, -6]

can be shortened to

SET5 ← [6, 4,..., -6]

This form can be extended to allow as many increment specifications as desired as long as the first two and the last elements of each group appear explicitly and the groups do not overlap, as in the following definitions:

SET6 ← [1, 2,..., 5, 7, 8, 10,..., 14]
SET7 ← [11, 10,..., 7, 5, 3,..., 0]

which are equivalent to:

SET6 ← [1, 2, 3, 4, 5, 7, 8, 10, 12, 14]
SET7 ← [11, 10, 9, 8, 7, 5, 3, 1]

Note that the last element of a group indicates the largest or smallest acceptable value for that group whether or not it is a member of the resulting set.

A further extension for 1-dimensional set definitions is the ability to specify the repetition of an element a fixed number of times. The format is similar to PL/1, for example,

SET8 ← [1(3), 4, 5(2)]

is equivalent to

SET8 ← [1, 1, 1, 4, 5, 5]

2.2.2.1.2 Comparison Sets

The comparison set is the result of comparing data values via some boolean expression. The general format is

$$\underline{\text{SET}}\ [J_1,\ldots,J_n:\ \langle\text{boolean expression}\rangle]$$

where each J_i is an identifier. The comparison set is specified to allow parallel generation of an index set at execution time. Thus at least one of the identifiers must be a subscript in the boolean expression and should also be under simultaneous control. The result of this definition is an n-dimensional set consisting of elements for which the boolean expression is true. For example, suppose A and B are vectors with elements

A: (-1, 3, 2, 10)

B: (2, -3, 1, 12)

and SET9 is defined as

$$\underline{\text{FOR}}\ (I)\ \underline{\text{SIM}}\ ([4, 3, 2, 1])\ \underline{\text{DO}}$$

$$\text{SET9} \leftarrow \underline{\text{SET}}\ [I:\ A[I] < B\ [I]]$$

The resulting set consists of the elements 1 and 4. To illustrate further consider the following matrices and statement:

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 4 & 2 \\ 4 & 2 & 3 \end{pmatrix} \qquad B = \begin{pmatrix} 4 & 1 & 0 \\ 3 & -1 & 1 \\ 2 & 5 & 3 \end{pmatrix}$$

```
FOR (I, J) SIM ([1, 2, 3]×[1, 2, 3]) DO

SET10 ← SET [J, I:  A[I, J] < B[I, J]]
```

the resulting set is [[1, 1], [2, 1], [1, 2], [2, 3]]. These sets are designed for use under simultaneous control; however, in the case of the 1-dimensional set, the ordering is monotonically increasing if one desires to use it for sequential control. Also it should be noted that the above examples are small for clarification, as much larger data sets will typically be involved on ILLIAC IV.

2.2.2.1.3 Set Ordering Definition

One dimensional sets are ordered from left to right when their elements are explicitly stated in the basic or shortened list form or in monotonically increasing order as mentioned above for comparison sets. Sets of greater dimension cannot be used in loop control and thus their ordering is not a user concern.

2.2.2.1.4 Set Operations

Sets can be combined using operators such as concatenation and union. For example, the set of factorials {1, 2, 6, 24,...,N!} can be generated as follows:

```
TEMP ← 1;
SF ← [1];
FOR (I) SEQ ([2, 3,...,N]) DO
SF ← SF UNION [TEMP ← I × TEMP]
```

where UNION joins the new element to SF and orders SF so that it is monotonically increasing. The use of concatenation (CONCAT) allows the user to add elements to either end of a set without a reordering by the compiler and REVERSE specifies the reverse ordering of a 1-dimensional set. The cartesian product (×) and pair (,) operators are useful in set assignment statements and control statements. For example:

$$S1 \leftarrow \lceil [1, 2, 3, 4] , [2, 4, 6, 8] \rceil$$
$$S2 \leftarrow \lceil [1, 2] \times [3, 4] \rceil$$
$$S3 \leftarrow \lceil [1, 2] \times [3, 4] , [5, 6] \rceil$$

are equivalent to:

$$S1 \leftarrow \lceil [1, 2], [2, 4], [3, 6], [4, 8] \rceil$$
$$S2 \leftarrow \lceil [1, 3], [1, 4], [2, 3], [2, 4] \rceil$$
$$S3 \leftarrow \lceil [1, 3, 5], [1, 4, 6], [2, 3, 5], [2, 4, 6] \rceil$$

where , has higher precedence than ×. Further set operations are discussed in the section of the TRANQUIL manual on set expressions [13].

2.2.2.2 Set Declarations

There are three types of set declarations, which are written in the following format:

{INCSET | MONOSET | GENSET | PATSET} $SET_1,\ldots,SET_m$

(<dimension number n>) $[L_1:U_1,L_2:U_2,\ldots,L_n:U_n,MS]$

where {} means one of the alternatives separated by | is chosen, L_i and U_i (i = 1,...,n) stand for the lower and upper bounds of component i, MS is the maximum number of elements or n-tuples possible, and SET_i (i = 1...,m) are set identifiers. INCSET, MONOSET, and GENSET signify 1-dimensional sets with the dimension number specification optional. The lower and upper bounds and size can be arithmetic expressions whose values are known at the beginning of the block of the associated set definition. The attribute INCSET (increment set) designates sets defined only in the form

first element, second element,...,last element

The attribute MONOSET (monotonic set) refers to sets that are strictly monotonic, the attribute GENSET (general set) refers to 1-dimensional sets, with no restrictions and PATSET (pattern set) refers to multi-dimensional sets (n > 1). The lower bound is 1 if not specified. Some examples are:

```
MONOSET   S1, S2 (1) [250,50]
PATSET    S3 (2) [0:100,0:200,30]
GENSET    S5 [50]
```

2.2.3 Sequential Control: The SEQ Statement

Sequential control is another term for loop control and is specified in the general form:

FOR (I_1,...,I_n) SEQ (II_1 {× | ,}...{× | , } II_n)

{DO | WHILE < boolean expression > DO} S

where the scope is the statement S, n as an integer, $I_i (i = 1,...,n)$ are control variables, and $II_i (i = 1,...,n)$ are 1-dimensional set identifiers.* The use of this statement is illustrated by the following examples.

(a) FOR (I,J) SEQ ([1, 2,...,10], [5, 10,...,50]) DO

A [I] ← B[I + 1] + C[J]

is evaluated as

A[1] ← B[2] + C[5];

A[2] ← B[3] + C[10];

.....

A[10] ← B[11] + C[50]

Note that the comma denotes pairwise ordering for the control variable values.

(b) FOR (I) SEQ ([2, 4, 6]) WHILE I < A [I] DO

A[I] ← B[I] - A[I]

will continue looping until the boolean expression is false or the index set has been exhausted. As in ALGOL no pass through the loop is made if the boolean expression is false after the index variable is assigned the initial value of 2.

(c) FOR (I, J) SEQ ([1, 2, 3, 4], [5, 6]) DO

B[I, J] ← A[I, J]

---

* In the remaining examples set definitions are also used for simplicity.

In this case the difference in size of the two defined sets is resolved by considering only the pairs (1, 5) and (2, 6), that is, the exhaustion of the smallest index set signals the end of the loop. To indicate otherwise an asterisk is placed after the set the exhaustion of whose elements is to be used as the stopping condition. This means that any other sets which run out of elements before completion will be repeatedly used as many times as necessary. If the previous statement is rewritten as

```
FOR (I, J) SEQ ([1, 2, 3, 4]*, [5, 6]) DO
        B[I, J] ← A[I, J]
```

the result is

```
B[1, 5] ← A[1, 5];
B[2, 6] ← A[2, 6];
B[3, 5] ← A[3, 5];
B[4, 6] ← A[4, 6];
```

(d)
```
FOR (I, J) SEQ ([1, 2] × [6, 7, 8]) DO
        A[I, J] ← B[J, I];
```

yields

```
A[1, 6] ← B[6, 1];
A[1, 7] ← B[7, 1];
A[1, 8] ← B[8, 1];
A[2, 6] ← B[6, 2];
A[2, 7] ← B[7, 2];
A[2, 8] ← B[8, 2];
```

where the lengths of the two sets do not create the problem that occurred with the pairwise operator. This example also illustrates that the frequency of element change is greatest for the rightmost set used.

(e) FOR (I, J, K) SEQ ([1, 2], [3, 4, 5] × [6, 7]) DO

C[I, K] ← D[J, K]

yields

C[1, 6] ← D[3, 6];

C[1, 7] ← D[3, 7];

C[2, 6] ← D[4, 6];

C[2, 7] ← D[4, 7]

where the comma has higher precedence.

For exits from loops via any means the control variables retain the last value assigned.

2.2.4 Simultaneous Control: The SIM Function and the SIM Statement

The parallel structure of ILLIAC IV is utilized through TRANQUIL by the specification of a simultaneous control function or statement. The functional form (refer to Appendix A for notation) is:

SIM BEGIN

list < assignment statement > separator;

END

where the enclosed assignment statements are executed simultaneously; i.e., the data used by any one of them is the data available before the SIM function was encountered. In the following example, the

contents of vectors A and B are interchanged without specification of a temporary location:

```
SIM BEGIN    A ← B;
             B ← A    END
```

The general statement form for simultaneous control is:

$$\underline{\text{FOR}}\ (I_1,\ldots,I_n)\ \underline{\text{SIM}}\ (II_1\ \{\times\ |,\ \}\ldots\{\times\ |,\ \}\ II_m)\ \underline{\text{DO}}\ S$$

where m, n are integers, I_i (i=1,...,n) are control variables, II_i (i=1,...,m) are k-dimensional sets ($0 \leq k \leq 7$), n equals the total number of dimensions of all II_i, and S is a statement. For this statement each substatement S_i of S is executed with the data available before it is reached, i.e., just as if a SIM function was placed around each S_i. In this regard it is important to note that simultaneous control is not loop control, but designates that each S_i is to be executed in parallel and thus the order of the associated sets is not important.

Several examples will illustrate the use of simultaneous control.

(a)
```
FOR (I, J) SIM ([1, 2, 3]*, [4, 5]) DO
                A[I, J] ← B[J, I]
```

is evaluated as

```
SIM BEGIN   A[1, 4] ← B[4, 1];
            A[2, 5] ← B[5, 2];
            A[3, 4] ← B[4, 3]
    END
```

(b) FOR (I) SIM ([1, 2,...,256]) DO

A[I] ← A[I-1]

yields

SIM BEGIN A[1] ← A[0];

A[2] ← A[1];

.....

A[256] ← A[255]

END

which shifts part of a vector one position.

(c) FOR (I, J) SIM ([1, 3], [2, 4]) DO

BEGIN A[I] ← B[J];

A[J] ← 3

END

yields

SIM BEGIN A[1] ← B[2];

A[3] ← B[4]

END;

SIM BEGIN A[2] ← 3;

A[4] ← 3

END

(d) FOR (I, J) SIM (II, JJ) DO

BEGIN

C[I, J] ← 0;

FOR (K) SEQ (KK) DO

C[I, J] ← C[I, J] + A[I, K] × B[K, J];

END

is a general routine for the multiplication of two matrices A and B.

It should be noted again that when a set is used in a sequential or simultaneous control statement, it cannot be altered in the scope of that statement.

2.2.5 Nested SEQ and SIM Statements

The sequential and simultaneous control statements described in the previous two sections can be nested. This nesting is straightforward except in the case where a SIM statement occurs within the scope of another SIM statement. When this occurs the area common to them is executed simultaneously with their sets related by the cross product operator as illustrated in the following example:

```
FOR (I, J) SIM (II, JJ) DO BEGIN
FOR (K) SIM (KK) DO BEGIN
        ... area A ....
END;
        .........
END
```

where in area A the control statement in effect is

FOR (I, J, K) SIM (II, JJ × KK) DO

This example is readily extended to nesting in greater depth.

2.2.6 If Clauses

General forms:

(a) IF {FOR (I_1,...,I_n) SIM (II_1{× | , }...{× | , } II_m)
| <empty>} {ANY | ALL | <empty>} |
<boolean expression> THEN

(b) IFSET <indexed boolean expression> THEN

Form (a) of the if clause can appear in designational, arithmetic, boolean, and set expressions. It results in a single logical value. The common form

IF <boolean expression> THEN

is easily seen to be a subclass of (a). The ANY and ALL modifiers are used when the boolean expression involves at least one subscript under SIM control and results in a true value when any or all of the tests are successful, respectively. For example, if the vector A of length 2 has elements 5 and 10:

IF FOR (I) SIM ([1, 2]) ANY A[I]< 7 THEN

has the value true since A[1]< 7. When the SIM appears in the if clause its scope extends only over the boolean expression.

Form (b) can be used in arithmetic and boolean expressions of assignment statements where the boolean expression of the if clause and the left hand part of such a statement both have subscripts under SIM control. The expression after the THEN is executed as if the set under SIM associated with the common subscript contains only those original elements for which the test is true and after the ELSE only those for which the test is false. Thus a single logical jump does not occur; rather, the original index set is reduced under

SIM control via the boolean test for use after the THEN and the ELSE when it occurs. Further, if this set is paired with any other set then only those pairs for which the test is true for the original set are used after the THEN and similarly after the ELSE for the false test. For example, if A[I, J] = 2 I + J,

```
FOR (I, J) SIM ([1, 2, 3, 4], [5, 6, 7, 8]) DO
    A[I, J] ← IFSET  A[I, J]< 8 OR  A[I, J]> 10 THEN
                        20 ELSE 30
```

yields

```
A[1, 5] ← 20;
A[2, 6] ← 20;
A[3, 7] ← 30;
A[4, 8] ← 20
```

Form (b) also can be used in an IF statement where similar action is taken. An example of both the first and second cases is

```
FOR (I) SIM ([1, 2,...,100]) DO
    T[I] ← IFSET A[I] < B[I] THEN A[I] ELSE B[I]
```

is equivalent to

```
FOR (I) SIM ([1, 2,...,100]) DO
    IFSET A[I] < B[I] THEN  T[I] ← A[I]
                      ELSE  T[I] ← B[I]
```

In either form T[I] ← A[I] for all values of I for which the boolean expression is true and T[I] ← B[I] otherwise.

When IF statements are nested the innermost THEN and the immediately following ELSE will be treated as one pair, and from this center the pairs proceed outwards. However, any desired arrangement is possible using BEGIN and END in forming compound statements.

2.2.7 Label and Switch Declarations

Labels are identifiers which must be declared before their use in statements and switch declarations. The declaration for a label is:

$$\underline{\text{LABEL}}\ L_1,\ldots,L_n$$

where L_i is a label identifier. Labels may be declared locally or globally with respect to the block in which they appear.

A switch must be declared before it is used, either in statements or in other switch declarations. The switch identifier in a declaration is followed by a list of designational expressions.

Examples:

$$\underline{\text{SWITCH}}\ \text{SWT1} \leftarrow L_1, L_2, L_3;$$

$$\underline{\text{SWITCH}}\ \text{SWT2} \leftarrow L_4, \text{SWT1}\ [\text{I}], \underline{\text{IF}}$$

$$\text{V} > 5\ \underline{\text{THEN}}\ L_5\ \underline{\text{ELSE}}\ L_6$$

where I and V are simple variables. The designational expressions are numbered from left to right starting at 1. They are evaluated each time they are selected. If another switch is involved its subscript value must be a positive integer and not exceed the number of designational expressions in that list.

2.2.8 Designational Expressions and GO TO Statements

Designational expressions in TRANQUIL involve labels, switches, and further designational expressions as in ALGOL. They are used in switch declarations and in the following examples in GO TO statements:

```
GO TO  FINISH
GO TO  SWT [2]
GO TO  IF A < 3 THEN CONTINUE ELSE STOP
```

where FINISH, SWT, CONTINUE, and STOP are labels, SWT is a switch, and A is a simple variable. Note that when designational expressions involve if clauses a single logical value must result. Also it should be noted that the destination for a go to statement must not be an interior block.

2.2.9 Statements and Block Structure

The block structure of TRANQUIL is identical to that of ALGOL. Hence, the following form is used:

```
BEGIN
     D; D;....D;
     S; S;....S;
END
```

where D represents a declaration and S a statement.

2.3 Comparison with Other Languages

2.3.1 ALGOL

TRANQUIL has the same general features of ALGOL; i.e., blocks, compound statements, assignment statements, conditional statements, go to statements, and iterative statements [15]. The conditional statements have been expanded as discussed in Section 2.2.6. The ALGOL FOR statement appears as the TRANQUIL SEQ statement which has been restructured to incorporate the use of index sets. The SIM statement is an addition following the same format as the SEQ. Together with the SIM statement for parallel specification there are the additional arithmetic, logical and relational operators which can be used on arrays. Storage mapping functions necessary for proper array storage are a further addition.

2.3.2 MADCAP

MADCAP is a language developed to aid a programmer in specifying combinatorial problems [16]. This obviously involves the use of sets which are denoted either in "tabulated" form, for example,

S1 = {0, 3, 4, 5, 8, 13}

S2 = {r, s, 10, 12}

or in "standard description" form, for example,

S3 = {x: for i = 5 to 31, x = i}

S4 = {x: for n = 1, 2,...,$\sqrt{N}$, if N=0(mod n), then x = n}

Included with the set when it is introduced is the maximum range of its elements. Further set description can be obtained using the operations of complementation, union, intersection, subtraction, and symmetric subtraction. This is similar to TRANQUIL. Further, these sets can then be used in conditional clauses:

if $j \in J$,...

if $a = b$,...

if $a \supset b$,...

and for iterative statements:

for $j \in J$

In general the index sets are unordered and thus the iterative statements are not the same as the TRANQUIL SEQ statement for which sets are considered to be ordered. In this general sense a MADCAP iterative statement is similar to the TRANQUIL SIM statement.

2.3.3 APL

A set in APL [17], a mathematically oriented language, is considered to be ordered and is denoted as a vector in which no two elements have the same value. These sets can also be altered using the standard set operators; however, they are not used in the high level control statement fashion of MADCAP or TRANQUIL. Loops must be specified using tests and jumps and set values choosen by indexing through the set. On these sets (or vectors in general for

APL only) the operations of cartesian product, concatenation, and end around shift can be used as in TRANQUIL.

APL allows the selection of elements of a vector or an array via the use of a pattern of 1's and 0's, for example, if S ← 1, 2, 3, 4, 5, 6 and U ← 1, 0, 1, 1, 0, 1 then U/S is 1, 3, 4, 6. The vector U can be thought of in TRANQUIL terms as a mode word and the result U/S as a set which can be used for index values of an array. This, however, is only one interpretation in APL for U/S can just as easily be considered a vector.

It should be noted that U/S only has 4 values whereas S has 6. Thus, if this method of compression is used to select elements of a vector to be added to similarly placed elements in another and then the sum is to be stored back in S in the positions of the selected elements expansion or masking must be used. In this way one picks out certain elements for an operation and then has to worry about putting the results back in place. For example, let

A ← 1, 2, 3, 4, 5, 6

B ← 2, 4, 6, 8, 10, 12

U ← 0, 1, 0, 1, 0, 1

C ← U/A + U/B

A ← /A, U, C/

where the final result in A is 1, 6, 3, 12, 5, 18 and where /A, U, C/ is a mask operation such that

$A_i = A_i$ or C_i according as $U_i = 0$ or $U_i = 1$.

This could also be written in terms of a loop as mentioned earlier. In TRANQUIL the same operation appears as

```
FOR (I) SIM ([2, 4, 6]) DO
    A[I] ← A[I] + B[I]
```

In TRANQUIL implementation the set [2, 4, 6] would generally be used in mode form; i.e., 010101, where this mode pattern is used to disable the PEs containing the first, third, and fifth elements of A and B, and not to compress or expand a vector.

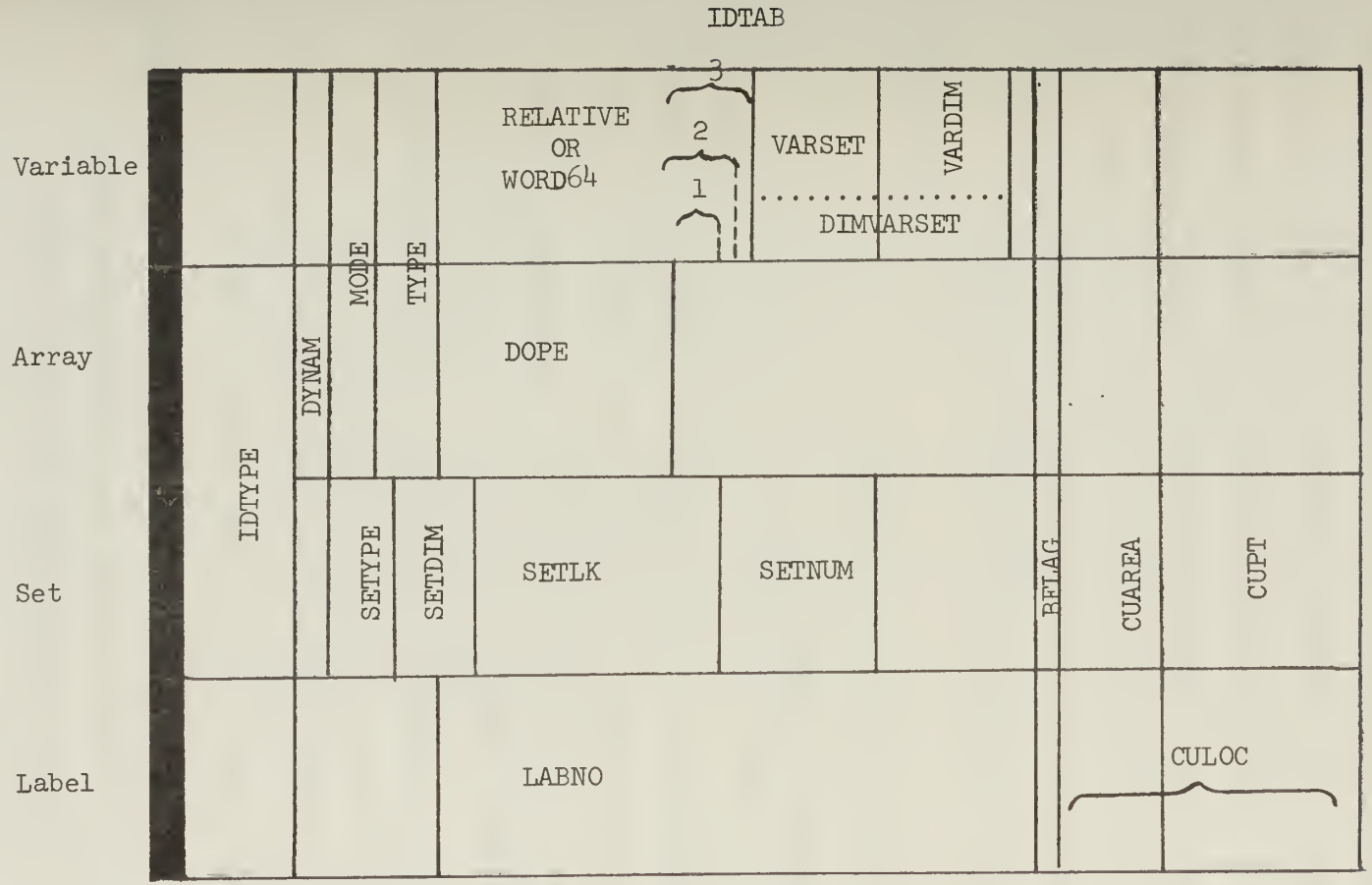
In comparing the case of using APL with TRANQUIL it should be noted that TRANQUIL is a higher level language than APL. This is of particular importance to users who do not desire to spend hours understanding the various ways that he can program sections of code and then deciding which way is best. Many of these decisions in TRANQUIL are left to the compiler; i.e., the sweep direction through an array being worked on under SIM control. In APL this decision is made by the programmer, in general.

3. COMPILER IMPLEMENTATION OF CONTROL STATEMENTS

3.1 Introduction

The syntax of TRANQUIL has been specified in a form which is accepted by the syntax preprocessor of the Translator Writing System (TWS) being built at the University of Illinois. The preprocessor automatically generates the parsing algorithm for the compiler. In pass 1 of the compiler the recognition of source code constructs invokes calls, via the action numbers embedded in the syntax definition, to semantic actions. These actions build descriptor tables containing, in part, information about declaration types, attributes, and block structure, and transform the source code into an intermediate language form which is composed of operators and operands, the latter being references to descriptor tables.

One table of importance is the identifier table (IDTAB). This table contains information about declared variables, arrays, sets, and labels. These entries have associated fields which are shown in Figure 1. A location in IDTAB is reserved when a declaration is encountered and thus all declarations for a given block occur in one connected segment. The BFLAG field is used to denote whether the end of the scope of an identifier has been reached. The other fields will be described later or else are described in [13, 14]. For identifiers that are declared more than once



1. WORD 32
2. WORD 16
3. WORD 8

Figure 1. The identifier table

and where these declarations are nested the BLOCK stack is used to keep track of those not presently valid.

Pass 2 is the main body of the compiler. The intermediate language stream is read and operators call pass 2 semantic actions for generation of assembly language instructions using associated operands. This brief summary appears in a flow chart in Appendix E and serves as a springboard for implementation considerations. In the following section the efficient allocation and use of CU registers and the storage and use of sets are discussed.

3.2 The Allocation and Use of CU Storage

3.2.1 CU Structure and Instructions

The allocation and use of CU registers is an important ILLIAC IV problem since CU instructions which cannot be overlapped with PE instructions leave all PEs idle. The registers of particular importance are the 4 accumulator registers (CARs) and the 64 local data buffer registers (LDBs), each register having 64 bits. The CARs serve as accumulators for all CU arithmetic operations, as the source of PE memory addresses, as the source of the first operand for CU and other instructions requiring two operands, as a local memory location, and as an index register. The LDB serves as a 64 word random access memory. The basic CU instructions include shifts, CU register transfer instructions, increment-test-skip instructions, 24-bit 2's complement addition and subtraction, leading 1's or 0's detection, 1 and 8-word loads from PE memory, and

a 1-word store to PE memory. It is important for a storage allocation scheme composed of a set of procedures for the assignment of CU locations to the compiler writer that no 8-word store to memory is allowed. Thus a sharp emphasis in development of such procedures was placed on minimizing stores to memory and on interleaving these stores with PE instructions.

3.2.2 An Implemented Solution

The setting of the LDB for the CU handling procedures is the following:

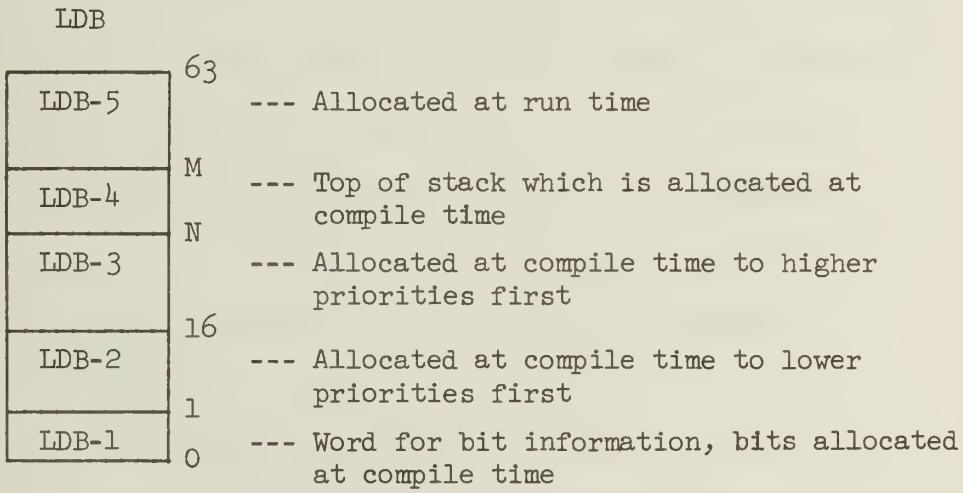


Figure 2. The division of the LDB

The concern of this paper is that portion of the LDB which can be allocated at compile time.

Underlying the allocation scheme for areas LDB-2 and LDB-3 is a set of priorities ranging from 0 to 3. Assignments to

LDB-2 are kept as much as possible to calls for an LDB register with priorities 0 or 1, while assignments to LDB-3 are kept as much as possible to calls for an LDB register with priorities 2 or 3. The reason for this division is to provide an area for 8-word loads from memory; i.e., the 16 word area LDB-2, where low priority assignments are usually made, and thus where the least penalty occurs in additional stores in making way for an 8-word block as well as the additional loads to return them to the CU. The lower priorities then are to be used for words that are used infrequently and the higher priorities for frequently needed information such as a present index value or a base address. These assignments are based on intuition and experience. CARs 1, 2, and 3 are also allocated via the priority specification. However, CAR 0 (LOOSECAR) is considered as free and can be used at anytime by the compiler writer.

In Figure 2, LDB-1 is an area reserved for the storage of bits of information, thus making it possible to store away a bit without using a whole word. The remaining area, LDB-4, contains the top of a stack (COMSTK) which extends into PE memory, It begins in location N and ends in M, where these locations can be adjusted easily by the compiler writer.*

---

* Further references to LDB registers will not refer to this area or LDB-1, but only to LDB-2 and LDB-3.

The following procedures are available for use in obtaining a CAR or LDB location:

INTEGER PROCEDURES	PROCEDURES
GETBIT	FREEBIT(BITNO)
CSPUSH(PRIORITY, PLACE, LOC)	CSPOP
GETCAR(PRIORITY, LINK, MOVECHANGE)	FREECAR(NUMBER)
GETLDB(PRIORITY, LINK, MOVECHANGE)	FREELDB(NUMBER)
	MOVETOMEM(LINK)
	MOVETOCS(PLACE, LOC)

A system of pointers is used in keeping track of what registers are in use and what other locations they are associated with. The pointer system appears as:

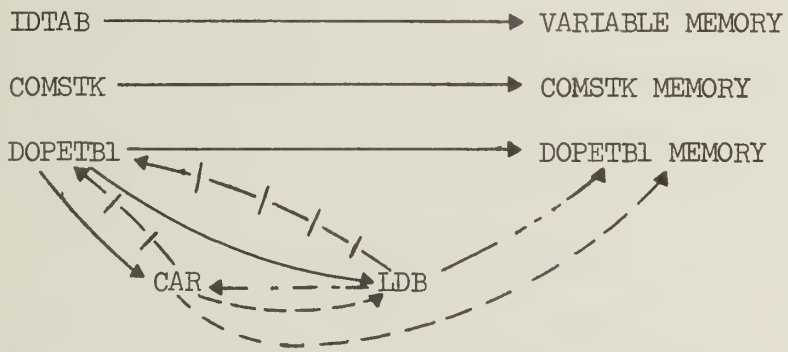


Figure 3. CU allocation pointer system

The 3 tables IDTAB, COMSTK, and DOPETBl each contain a field (CULOC) composed of an area designation (CUAREA) and a location indicator (CUPT) that points to an associated CAR, LDB, or

corresponding memory location in that order.* The IDTAB positions used are restricted at present to those associated with variables. The general parameters are the PRIORITY, LINK, and MOVECHANGE parameters. The LINK is composed of 2 fields:

TABLK

The TABSTK field refers to one of the tables mentioned above and the TABSTKPT to a location in the table designated. The MOVECHANGE parameter is used to indicate information about the word to be associated with the assigned location, and can have the attributes: CURESET, CUSET, CUMOVE, CUSETMOVE, or MOVECOPY. CUSET and CURESET are used to indicate that a register contains new or old information, respectively, this information being stored in a bit called the CUCHANGE bit. New information will then be stored properly when its associated location is reassigned or freed. The attributes CUMOVE, CUSETMOVE, and MOVECOPY produce code to move the word pointed at by the link to the assigned location. Further, CUMOVE acts like CURESET and CUSETMOVE like CUSET. MOVECOPY indicates

---

* For further information on DOPETBl and IDTAB see [14]. The COMSTK is the mock-up table of the stack associated with the area LDB-4.

that if the move is between a CAR and an LDB location the priority and SET or RESET information are also to be carried along.

3.2.3 Use of the Procedures

3.2.3.1 INTEGER PROCEDURE GETLDB(PRIORITY, LINK, MOVECHANGE)

This procedure selects an LDB location from LDB-2 or LDB-3. Furthermore,

1) if the linked word is associated with an LDB location and not with a CAR the CUCHANGE bit is set from the MOVECHANGE parameter, and the associated integer returned is the related location number.
2) If the linked word is associated with an LDB register and a CAR register then the CAR is freed, and the LDB location returned.
3) If priority 0 or 1 is specified when 1) and 2) do not occur the following actions are taken till a location is found:
    a) use a free space in LDB-2,
    b) use a free space in LDB-3,
    c) remove a word from LDB-2 in the lowest possible priority group and use the vacated location,
    d) remove a word from LDB-3 in the lowest possible priority group and use the vacated location

or if priority 2 or 3 is specified when 1) or 2) do not occur the following actions are taken till a location is found:

a) use a free space in LDB-3,

b) use a free space in LDB-2,

c) remove a word from LDB-3 in the lowest possible priority group and use the vacated location,

d) remove a word from LDB-2 in the lowest possible priority group and use the vacated location.

4) If the LINK parameter is zero an LDB location is gotten without pointer connections, the PRIORITY parameter used in choosing a location as in 3) and the MOVECHANGE parameter ignored. Care must be taken in using a zero LINK since unless the returned location is frozen its contents could disappear if another GETLDB or a GETCAR call is made.

5) An LDB location can be frozen, i.e., removed from possible allocation, by

```
FREEZELDBNUM:= number ;
FREEZELDB
```

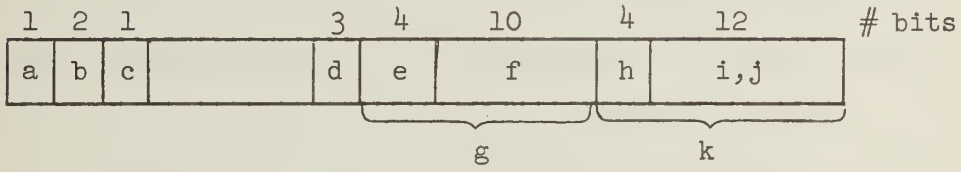
and it can be unfrozen by

```
FREEZELDBNUM:= number ;
UNFREEZELDB
```

6) This procedure uses LOOSECAR.

The implementation of LDB allocation utilizes a mock-up of the LDB called CULDB and several fields from the COMSTK mock-up. The fields are:

CULDB location



COMSTK location

a: CUCHANGE
b: CUPR
c: FREEZEBIT
d: CARLK
e: BACKLKAREA
f: BACKLKPT
g: BACKLK
h: TABSTK
i: TABSTKPT
j: FLDBLK
k: TABLK
l: PRLKUP
m: PRLKDN

The CUCHANGE bit is used to indicate whether the contents of the word are considered new or old(1 or 0, respectively). The CUPR field contains the priority and the FREEZEBIT indicates whether or not an LDB location is frozen. If there is an associated CAR the CARLK contains this number (— · — · →).* The BACKLKAREA points to

VARIABLE, COMSTK, or DOPETBl memory with location BACKLKPT (—— ... —→). TABSTK points to IDTAB, COMSTK, or DOPETBl with location TABSTKPT (—| —| —→). The FLDBLK is used to keep 2 linked lists of free LDB locations, one for LDB-2 and the other for LDB-3. The array USERPRPT [0:7] is a user priority pointer array, LDB-2 priorities with 0-3 indices and LDB-3 priorities with 4-7 indices. The allocation and freeing of LDB locations does not in general, take place on a stack basis and priority lists for LDB locations are thus necessary in choosing a word to leave the LDB. The PRLKUP and PRLKDN fields are used for the up and down pointers of this list.

3.2.3.2 INTEGER PROCEDURE GETCAR(PRIORITY, LINK, MOVECHANGE)

This procedure selects a CAR register on a priority basis. Further,

1) CAR 0, known as the LOOSECAR, is selected only when USELOOSECAR is TRUE. This CAR is considered by all allocation routines as being free whether it is linked or not.
2) The PRIORITY and LINK parameters are as in GETLDB.
3) If the linked word is in a CAR that CAR number is

---

* Arrows in parenthesis refer to Figure 3.

returned and the CUCHANGE bit is set from the MOVECHANGE parameter.

4) If the LINK parameter is zero a CAR is selected without pointer connections, the PRIORITY parameter used in choosing the CAR and the MOVECHANGE parameter ignored. Care must be taken in using a zero LINK since unless the returned location is frozen its contents could disappear if another GETCAR is used.

5) CARs 1, 2, or 3 can be frozen if they have a priority of 3, i.e., removed from possible allocation, by

    FREEZECARNUM:= number ;

    FREEZECAR

and they can be unfrozen by

    FREEZECARNUM:= number ;

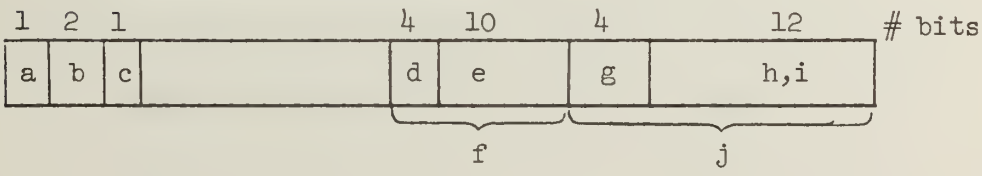
    UNFREEZECAR

Only two CARs can be frozen at any time.

6) This procedure uses LOOSECAR.

The implementation of CAR allocations utilizes a mock-up of the CAR registers having the following fields:

CAR register



a: CUCHANGE
b: CUPR
c: FREEZEBIT
d: BACKLKAREA
e: BACKLKPT
f: BACKLK
g: TABSTK
h: TABSTKPT
i: FCARLK
j: TABLK

The CUCHANGE, CUPR, FREEZEBIT, and TABLK fields are as in the LDB mock-up. The BACKLK field (- - - - - →) has the added possibility of an LDB register. The FCARLK field is used to keep a list of free CARs.

3.2.3.3 INTEGER PROCEDURE CSPUSH(PRIORITY, PLACE, LOC)

This procedure pushes a word onto the compile-time allocated stack (COMSTK). Further,

1) the integer result is the LDB location in LDB-4 that is allocated.
2) The PLACE parameter indicates whether a CAR or LDB location is to be associated with the top of the stack when CARNUM or LDBNUM are used. When this occurs the CAR or LDB location CUCHANGE bit is set on to indicate a new value and the priority is set. Otherwise CSNUM should be used if just a location is desired.
3) The LOC parameter indicates the place in the CAR or LDB for 2) and if CSNUM is used is ignored. For example,

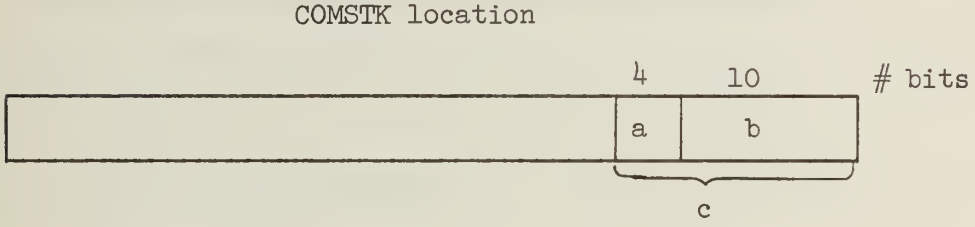
TEMP:= GETCAR(0,0,0);

LOC:= CSPUSH(2,CARNUM,TEMP)

results in the selection of a CAR that is associated with the top of the COMSTK and which has priority 2.

4) No moving is done and thus LOOSECAR is not used.

Implementation involves a mock-up COMSTK:



a: CUAREA

b: CUPT

c: CULOC

where CULOC is as described earlier. When LDB-4 is full the lowest element in the stack is moved to the stack extension in memory.

3.2.3.4 PROCEDURE CSPOP

This procedure pops a word from COMSTK with the following possible actions:

1) If the top of the stack is associated with a CAR and/or LDB register, these registers are freed.

2) If more than 1 location in LDB-4 is available stack words in PE memory are loaded into LDB-4 until 1 word in LDB-4 is free or until no words are left

in PE memory.

3) This procedure uses LOOSECAR.

3.2.3.5 PROCEDURE MOVETOCS(PLACE, LOC)

This procedure moves a word in a CAR register or LDB register in area LDB-2 or LDB-3 to the appropriate area, LDB-4 or stack memory. Further,

1) all associated LDB or CAR registers are freed.

2) The move occurs only if the CUCHANGE bit of the CAR or LDB location given by the parameters is on.

3) This procedure uses LOOSECAR.

3.2.3.6 PROCEDURE MOVETOMEM(LINK)

This procedure moves a word in a CAR or LDB register in areaLDB-2 or LDB-3 to the appropriate memory, i.e., variable memory, stack memory, or dope table memory. Further,

1) the LINK points to IDTAB, COMSTK, or DOPETBl as before. If the CULOC field in the designated table points to its corresponding memory location or the CAR or LDB location has a zero CUCHANGE bit no move occurs.

2) The associated LDB and/or CAR registers are freed.

3) This procedure uses LOOSECAR.

3.2.3.7 FREELDB(NUMBER)

This procedure frees the LDB register whose number is given by the parameter NUMBER. An associated CAR, if it exists, is also freed and the freed LDB number goes on the FLDBLK list.

3.2.3.8 FREECAR(NUMBER)

This procedure frees the CAR register whose number is given by the parameter NUMBER. If there is an associated LDB register, its CAR link (CARLK) is severed.

3.2.3.9 INTEGER PROCEDURE GETBIT

This procedure allocates a bit in LDB [0].

3.2.3.10 INTEGER PROCEDURE FREEBIT(BITNO)

This procedure frees the bit given by the parameter BITNO.

3.3 The Storage and Use of Sets

3.3.1 Introduction

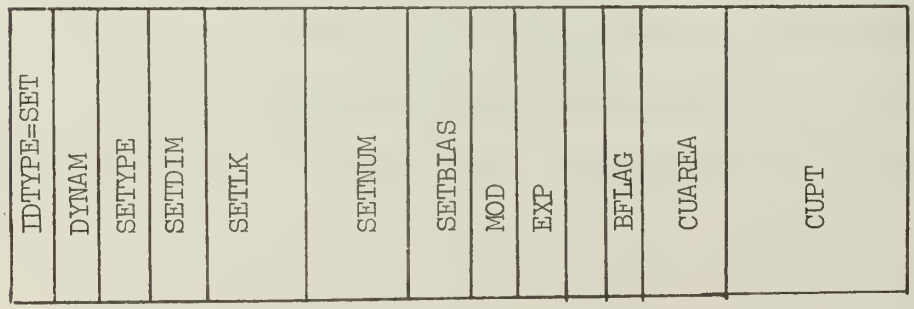
Associated with the introduction of sets in TRANQUIL is the task of finding storage schemes which can be used efficiently. The storage schemes must take into account that sets can be used for loop control, for enabling or disabling PEs and for PE indexing. This requires the possibility of representing sets via the use of mode words or explicit numbers. Mode words are 64-bit words used

to store the elements of a set by position number, where a 1 in the n-th bit indicates that n corresponds to an element of the set. Further required is the ability to change from one of these forms to the other as the use of a set changes. The decision on a storage scheme must also account for the fact that the order of execution of a program, in general, depends on some run time calculations. This requires keeping track of the present set elements associated with a particular set identifier. Finally, it must allow the reservation of memory space for a set within the block in which the set is declared. This is apparent when one notes that the storage form(s) needed for a set expression associated with a SIM control statement cannot be determined, in general, until the arithmetic or boolean expressions or statements in the scope of the SIM statement have been analyzed in pass 2.

3.3.2 Set Storage Tables

The key location for set information is found in the identifier table, in a word having the following fields:

Set IDTAB location

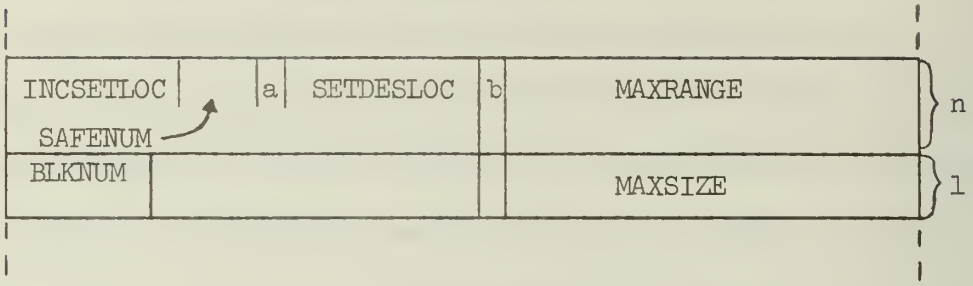


1) The IDTYPE field contains the SET indicator.
2) The SETYPE field contains the declaration type.
3) The SETDIM field contains the dimension of the set.
4) The DYNAM field indicates whether the range and size specifications are given by constants or general arithmetic expressions.
5) The SETNUM field contains a distinct number for every declared increment set starting at zero and a distinct number for every declared monotonic or general set also starting at zero.
6) The SETBIAS field is used to denote whether a set is known to have any negative numbers or not.
7) The MOD field indicates if the mode form of storage is desired.
8) The EXP field indicates if the explicit form of storage is desired.
9) The SETLK fields points to further set information in the table called SETTAB.

The first 5 fields explained are filled in during the first pass and the remainder are set and used in pass 2 of the compiler.

The table SETTAB is an extension of IDTAB for sets and appears as:

SETTAB information



One of the important details in a set storage scheme is the reservation of ILLIAC IV memory space. When the maximum range(s) and size are constants the compiler can fill in the MAXRANGE field for each of the n dimensions and the MAXSIZE field. This is the only case considered here. Such information allows compile time memory allocation and, further, set density calculations can be made to help determine which way to sweep through an array whose indices are under SIM control. The field labeled (b) in SETTAB is a presence bit for the MAXRANGE and MAXSIZE fields.

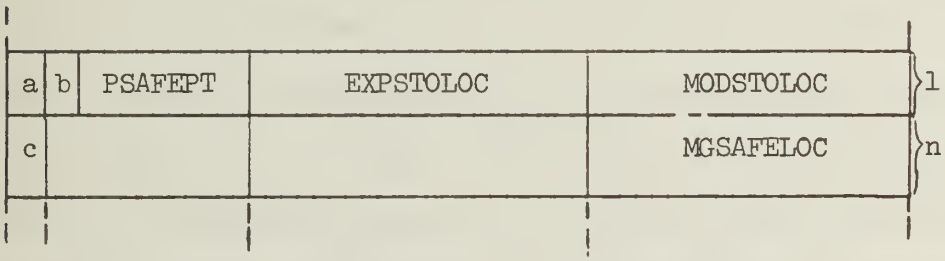
3.3.3 A Set Storage Scheme

One solution to the memory allocation problem in TRANQUIL requires that at the beginning of a block memory space is allocated in the set storage area (SETSTO) according as to whether a set is declared monotonic or general, in the former case enough space for mode words and in the latter for explicit numbers. Thus, in the program of Appendix D space for the mode word representation of the sets II, KK, and LL are allocated at the beginning of the block in which they are declared. The arithmetic statement in line 19

requires LL in mode form and JJ in explicit form. This then requires the translation of mode words into explicit numbers and a place to put these numbers in SETSTO. Space is allocated when this is known, after mode word space has been allocated for LL. In order to accomplish this an in-use linked list of spaces for each block is kept along with a free list of spaces available for allocation. Additions to the use list are made at block heads and interior points and additions to the free list are made at block ends.

The bookkeeping concerning what set is presently valid is carried on in the run-time set descriptor table (SETDES) which for a particular set looks like:

SETDES information



The fields EXPSTOLOC and MODSTOLOC in the first word contain the first location in the SETSTO area for the explicit and mode word representations of a set and fields a and b are their presense bits, respectively.

The set storage scheme is amplified so that set definitions involving lists of arithmetic expressions exemplified by

[1, 3, -2, 6]

[1(2), 3(6)]

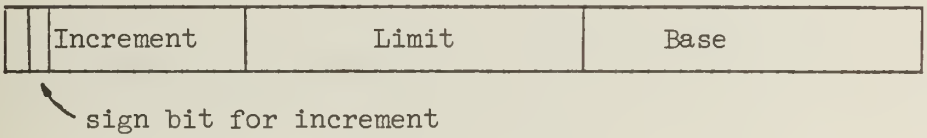
can be stored permanently. The area for their storage is called the monotonic-general set area (MGSAFE). Sets stored here are used directly. All other set expressions for non-dynamic sets are allocated space in SETSTO.

In SETDES, then, n words contain the addresses of the n set definitions in MGSAFE in the field MGSAFELOC with the 1-st word field PSAFEPT indicating which one of these is valid at any point in program execution, if any. Field c is the MGSAFELOC presense bit for compile-time use.

The address in SETDES of a set is found in the SETDESLOC of SETTAB where field a is a presense bit for this location. Also in SETTAB the SAFENUM field contains the number of set definitions to be stored in the MGSAFE area and the INCSETLOC field points to a 2 word block where an increment specification is stored as outlined in the next section. The BLKNUM field in SETTAB allows allocation of space in SETTAB to be associated with the block in which the set is declared. The following section deals with the specific storage representations for 1-dimensional sets in increment, mode, and explicit forms.

3.3.4 Increment, Mode, and Explicit Storage Forms and Their Use

The increment set (by declaration) is stored using two words per set. One word contains the first element, the increment, and the limit packed for use by CAR instructions:



The other word is a bias value based on zero and is used in the case where negative elements are, or may be, in the set. The base location and table name for these sets is INCSET and for a given set the appropriate two-word block is obtained by multiplying the value in the SETNUM field of IDTAB by 2 and adding the location number INCSET.

When an increment set is used for sequential control, CU test and increment instructions operate on the appropriate CAR register. When an increment set is used for simultaneous control, it can be expanded at run time into mode words or explicit numbers and stored in SETSTO. Mode words can be generated from an increment set by using a memory row that contains the PE numbers in ascending and descending order and regular mode patterns similar to the 4 PE system shown in Figure 4. In the figure the mode pattern was formed by considering the b_0 bits to be all ones, the b_1 bits to be alternating zeroes and ones, the b_2 bits to be two zeroes alternating with a one, and finally the b_3 bits to be three zeros alternating with a one. In the general 64 PE case, the word in the i-th

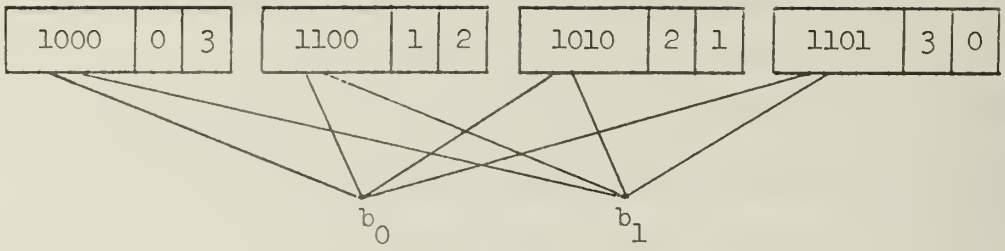


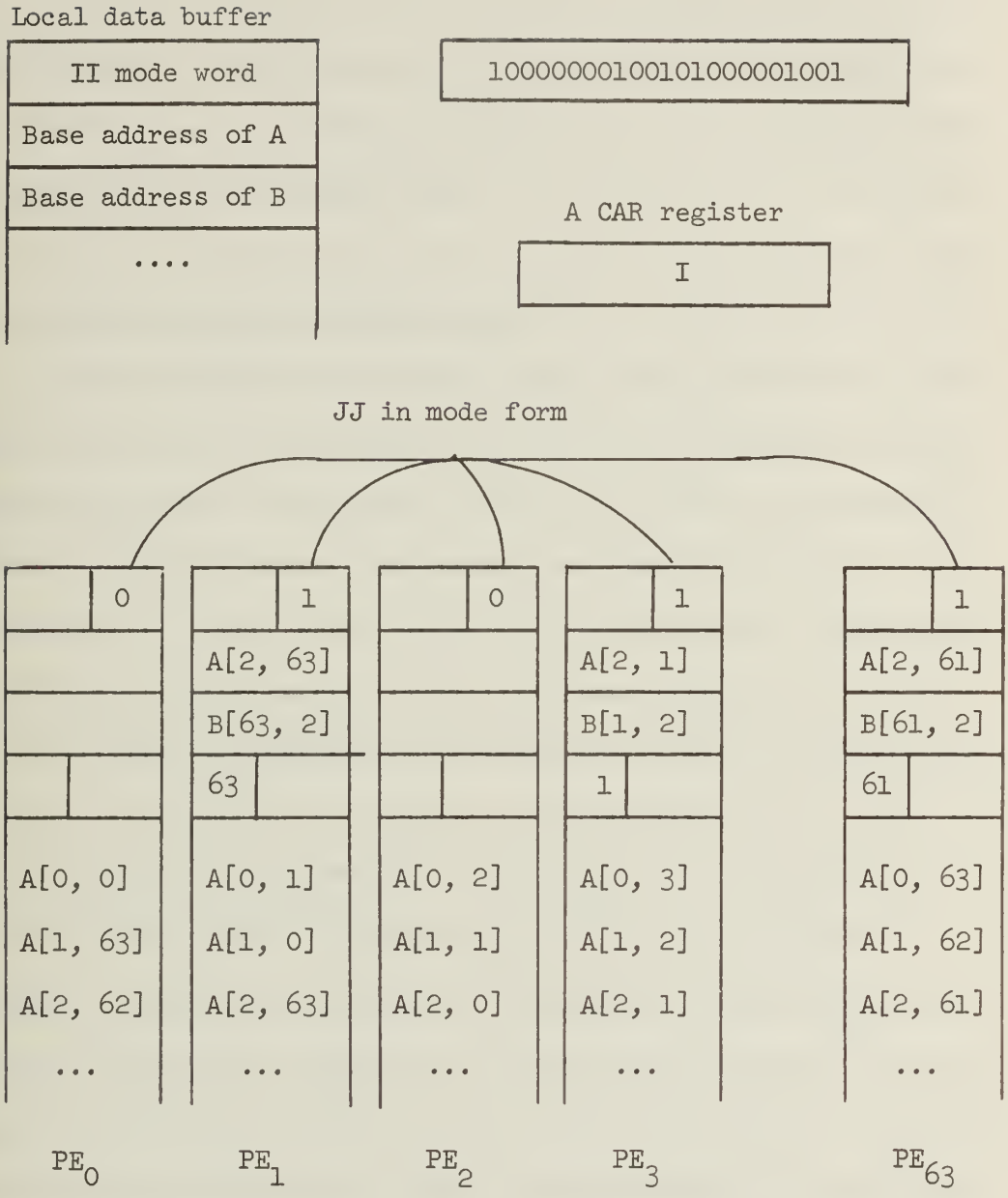
Figure 4. Mode patterns and explicit values for increment sets

PE is

Mode Pattern	i	63-i

where the 32-bit mode pattern results by considering the b_0 bits to be all ones; i.e., all PEs on when in use, the b_1 bits having the pattern 0101..., the b_2 bits 001001..., and the b_i bits 0_i10_i1 where 0_i stands for i zeroes. Explicit numbers can be generated using appropriate multiples of the PE numbers plus an addition or subtraction.

Now consider the example of expanding the increment set JJ of Appendix D into mode words and explicit numbers. The set JJ is used simultaneously in forming the comparison set KK by a boolean test on the skewed arrays A and B. For a given I every other element of A, as signified by JJ, is moved to the A register in the PEs using the base address of A that has been brought to the CU data buffer as indicated in Figure 5. Every other element



A refers to PE registers A.

B refers to PE registers B.

X refers to the PE index registers.

Figure 5. General PE and CU picture for line 10 of the problem in Appendix D where I = 2

corresponds to the b_1 bit mode pattern, this pattern appearing in the PE mode positions of Figure 5. The case for I = 2 is shown. Every other element of the I-th column of B is moved to the B register. In this case every other ascending PE number is used in the PE index registers (X) with appropriate routing to account for the skewed storage. For I = 2 in Figure 5, an end around route of two is necessary. Every other element of a column is fetched to the B register again by the use of JJ in mode form. For the explicit representation of JJ to be used in line 19 the ascending PE numbers are multiplied by 2 and then 2 is added to each of these resulting numbers.

The monotonic set is stored originally as a list of mode words together with a header word in the following form:

Bias	a	1	Number of Words

The bias is the first element of the set and the field a indicates whether the set is increasing or decreasing. Anytime mode storage is used this header is used. Similarly for general sets or explicit represention, a header word is used, but this time with the CU increment format where the base and increment are 1 and the limit is the number of elements at present. When either a MONOSET or GENSET is defined using the increment format, 2-word blocks of space are allocated in the INCSET table with the location appearing in the field INCSETLOC of SETTAB.

Size restrictions are imposed on the number of words used for mode or explicit storage for non-dynamic sets and if they are exceeded they are treated as dynamic sets. These restrictions are 64 and 512 words, respectively.

3.3.5 Index Set Usage in SEQs and SIMs

The routines for outputting code for SEQ loops involves 3 particular cases, when the set is stored in the increment, mode, and general forms. The increment form was discussed in the last section. The selection of loop values from mode forms is accomplished via leading ones detection with adjustments for the bias while for the explicit form the loop values are simply loaded into the CU from set storage. For example, the sets II and KK in Appendix D are examples of monotonic sets and are stored in mode form with no bias value in these particular cases. For looping on II the CU does leading ones detection on the II mode pattern, illustrated in Figure 5, to determine the explicit set elements used in the CAR as CU indices for array A, and KK is used in mode form in the PE mode registers under simultaneous control. The header words aid in this task in an obvious manner and the CU stack is used to store intermediate calculations. If a loop is not left via its natural completion adjustments are made to ensure the proper CU set-up.

In order to keep track of the control variables associated with the sets in a SEQ or SIM statement a variable stack (VARSTACK)

is used. This stack has the fields:

VARSTACK

The VARDIV field indicates the number of control variables in a SEQ or SIM statement. The SEQREL field is used in determining for a SEQ statement with more than one set whether there is SEQ within a SEQ or just a single SEQ. This is determined via whether the associated sets appear with the cross product or the pair operator, respectively. The VARLINK field points to the corresponding IDTAB position for the control variable.

For SIM control statements information is stored in a stack (VSSTACK) concerning the relationships between sets. The format is illustrated together with an example.

```
FOR (I,J) SIM (II,JJ) DO
  ......
  FOR (K) SEQ (KK) DO
    ......
    FOR (L,M) SIM (LL,MM) DO
      ......
      FOR (N) SIM (NN) DO
        ......
```

where all the sets are 1-dimensional results in:

VSSTACK

	1	0	0
	0	1	II
	2	1	JJ
	0	2	LL
	1	2	MM
	1	0	NN
	VSBLOCK	SETREL	SETADR

The VSBLOCK field indicates the number of VSSTACK entries per SIM via the non-zero integers where integers m greater than 1 indicate the occurence of m-1 SEQ statements. The SETREL field indicates the relations between the sets via groups. If there is a zero in this field the single set is considered a group while if some integer appears here the associated set belongs to that numbered group. Thus the pairs II,JJ and LL,MM form a group. The SETADR field contains the associated set address in IDTAB.

Returning to IDTAB, the control variables associated with SIM control have the VARSET field which contains a pointer to VSSTACK and the VARDIM field which contains the associated dimension. The VARLINK field is VARSTACK is used to zero these fields at the end of a SIM statement. The use of VSSTACK and these IDTAB fields is discussed in [13].

In closing this section is should be noted that little has been said about sets with dimension greater than 1, i.e., PATSETs. The reason for introducing these sets was to allow the user the ability to work on scattered elements of an array which is otherwise not possible. For example, the 2-dimensional mode pattern:

1 0 0 1
0 1 1 1
0 1 0 1
0 0 1 0

can be used to work on elements of the following array which are marked:

X			X
	X	X	X
	X		X
		X	

PATSETs are stored always in mode patterns which is possible since repeated elements are not permitted in defining these sets.

3.3.6 Mode Pattern-Explicit Word Conversion

The conversion of explicit numbers to mode patterns can be accomplished in parallel at run time by having each PE turn on the bit corresponding to the set number it has. For each 64-bit mode word the range of explicit elements is set at 64 and the mode bits are set only after the explicit numbers have been properly scaled.

The parallel run-time conversion of mode words into explicit numbers is done, however, less efficiently. This is

because only those PEs with mode bits on will contain set elements after using appropriately the PE numbers, thus leaving intervening PEs with no set elements. It is then required to pack these explicit numbers and as the number of gaps increases so does the number of routes and time to do this.

4. CONCLUSIONS AND SUGGESTIONS

There are a number of additions that should be considered for TRANQUIL with regard to control statements. These include the introduction of procedures and Burrough's case statements. It would also be useful to be able to specify the sweep direction through an array whose indices are under SIM control such as in line 22 of Appendix D. For example, to sweep by rows one could write

FOR (I*,K) SIM (II×KK) DO

Further, in such control statements the use of parentheses would allow the user greater power in grouping the sets after the word SEQ or SIM. Also with the addition of input-output statements should come the ability to read in sets as data.

The work that has been done with regard to sets has centered around 1-dimensional non-dynamic sets. Further work in the area of developing a scheme to handle dynamic sets is necessary. This is designed to be, in general, an extension to the facilities and tables presently available. The CU allocation schemes need to be expanded to allow for run-time space allocation. this again is designed to be an extension of what already exists.

APPENDIX A

METALANGUAGE FOR SPECIFYING SYNTAX

The TRANQUIL syntax in Appendix B is specified in a form of BNF which is extended as follows:

1) Kleene star:

$< A >^{*} = < > \mid < A > \mid < A > < A > \mid \ldots$

where $< >$ represents the empty symbol

2) Brooker and Morris' question mark (here &):

$< A > \& = < > \mid < A >$

3) List Facilities

list $< A > = < A > < A > *$

list $< A >$ separator $< B > = < A > [< B > < A >] *$

4) Brackets

$< T > ::= [< A > \mid < B > \mid < C >] < D >$

is equivalent to

$< T > ::= < R > < D >$

$< R > ::= < A > \mid < B > \mid < C >$

5) Metacharacters:

A sharp (#) must procede each of the following characters when they belong to syntactic definitions:

#, [,], *, ;, <, >.

In the syntax < * I > is used to designate an identifier and < * N > is used to designate a number. Further, the special words in the language are capitalized and underlined.

APPENDIX B

TRANQUIL SET AND CONTROL STATEMENT SYNTAX

B.1 Statements

< PROGRAM > ::= < BLOCK >

< BLOCK > ::= BEGIN list [< DECLARATION > # ;]
list < STATEMENT > separator # ;
[# ;] & END;

< STATEMENT > ::= < NONEMPTY STATEMENT > | < > ;

< NONEMPTY STATEMENT > ::= [< * I > : NOT =] *
[< CONTROL STATEMENT > |
GO TO < DESIGNATIONAL EXPRESSION > |
BEGIN < STATEMENT > END |
< BLOCK > |
< IF CLAUSE > < STATEMENT >
[ELSE < STATEMENT >] & |
< ASSIGNMENT STATEMENT >] ;

B.2 Control Statements

< CONTROL STATEMENT > ::= FOR (< SET VARIABLE LIST >)
[SEQ | SIM] (< SET NAME LIST >) DO < STATEMENT > |
FOR (< SET VARIABLE LIST >) SEQ (< SET NAME LIST >)
WHILE < BOOLEAN EXPRESSION > DO
< STATEMENT > | < SIM BLOCK > ;

< SET VARIABLE LIST > ::= list < * I > separator, ;

< SET NAME LIST > ::= list [< * I > | < SET DEFINITION TAIL >] [# *] & separator [, | ×] ;

< SIM BLOCK > ::= SIM BEGIN list < ASSIGNMENT STATEMENT > separator # ; [# ;] & END ;

B.3 Sets

B.3.1 Declarations

< DECLARATION > ::= {INCSET | MONOSET | GENSET | PATSET} list < SET SEGMENT > separator , ;

< SET SEGMENT > ::= list < * I > separator , [(< * N >)] & [# [list [< ARITHMETIC EXPRESSION > [: < ARITHMETIC EXPRESSION >] &] separator, #]] &

B.3.2 Definitions

< SET DEFINITION TAIL > ::= # [< LIST SET > #] | < COMPARISON SET > ;

< LIST SET > ::= <ELEMENT > [, < ELEMENT >,..., < ELEMENT > | , < ELEMENT >] * | < > ;

< ELEMENT > ::= #[list < ARITHMETIC EXPRESSION > separator , #] | < ARITHMETIC EXPRESSION > [(< ARITHMETIC EXPRESSION >)] & ;

< COMPARISON SET > ::= SET # [list < * I > separator , : < BOOLEAN EXPRESSION > #] ;

B.3.3 Set Assignment Statements

< ASSIGNMENT STATEMENT > ::=

< BOOLEAN ASSIGNMENT STATEMENT > |

< ARITHMETIC ASSIGNMENT STATEMENT > |

< SET ASSIGNMENT STATEMENT > ;

< SET ASSIGNMENT STATEMENT > := list [< SET IDENTIFIER >

[← | :=]] < SET EXPRESSION > ;

< SET EXPRESSION > ::= < SIMPLE SET > |

< IF CLAUSE > < SIMPLE SET > ELSE

< SIMPLE SET > ;

< SIMPLE SET > ::= < SET PAIR > [< SET PAIR >] * |

REVERSE < SET PAIR > ;

< SET PAIR > ::= < SET UNION > [PAIR < SET UNION >] * ;

< SET UNION > ::= < SET INTERSECTION > [[UNION |

DELETE | SMD | CONCAT] < SET INTERSECTION > * ;

< SET INTERSECTION > ::= < SET FACTOR > [INT < SET

FACTOR >] * ;

< SET FACTOR > ::= < SET OFFSET > [COM < SET OFFSET >] * ;

< SET OFFSET > ::= < SET PRIMARY > | < SET PRIMARY >

[+ | -] < ARITHMETIC EXPRESSION > ;

< SET PRIMARY > := < SET IDENTIFIER > | (< SET EXPRESSION <) |

CSHIFT (< SET EXPRESSION >,<ARITHMETIC EXPRESSION >) |

< SET DEFINITION TAIL > ;

B.4 Switch and Label Declarations

< DECLARATION > ::= LABEL list < * I > separator , |
SWITCH < * I > [← | : =] list
< DESIGNATIONAL EXPRESSION > separator , ;

B.5 Designational Expressions

< DESIGNATIONAL EXPRESSION > ::= < SIMPLE DESIGNATIONAL EXPRESSION > | < IF CLAUSE > < SIMPLE DESIGNATIONAL EXPRESSION > ELSE < DESIGNATIONAL EXPRESSION > ;

< SIMPLE DESIGNATIONAL EXPRESSION > ::= (< DESIGNATIONAL EXPRESSION >) | < SWITCH IDENTIFIER > # [< ARITHMETIC EXPRESSION > #] | < LABEL IDENTIFIER > ;

B.6 IF Clauses

< IF CLAUSE > ::= IF < CONTROL HEAD > & [ANY | ALL] &
< BOOLEAN EXPRESSION > THEN |
IFSET < BOOLEAN EXPRESSION > THEN ;

B.7 Global Operators

< ARITHMETIC EXPRESSION > ::= < GLOBAL PRIMARY > ;

< GLOBAL PRIMARY >::=[FOR (< SET VARIABLE LIST >)
SIM (< SET NAME LIST >)] & [SUM |
PROD | GOR | GAND | MAX | MIN]
< ARITHMETIC EXPRESSION >] ;

APPENDIX C

AN INTUITIVE LOOK ---
THE TRANSLATION OF FORTRAN INTO TRANQUIL

The translation of FORTRAN into TRANQUIL requires the detection of implicit parallelism in FORTRAN and its explicit representation in TRANQUIL. The explicit loops in FORTRAN such as

```
DO 12 I=1, 100
```

and the implicit loops involving the IF statement are possible indicators of potential parallelism.

C.1 Essential Ordering

The question immediately arises in the survey of loops in search for parallelism as to what the essential ordering of the statements enclosed is. If this can be determined unnecessary calculations can be removed from a loop and information is then available for optimizing code, for determining more efficient schemes for memory usage, and for parallel detection.

A possible method for determining the essential ordering is illustrated for a set of assignment statements involving variables:

A = B + C

B = D + E

E = A

A = B + D

C = D + F

Forming an assignment matrix

$T = [t_{ij}]$ where

t_{ij} = 1 if the variable j is on the right hand side, where i is on the left hand side

0 otherwise

yields

	A	B	E	A	C	D	F
A		1			1		
B			1			1	
E	1			1			
A		1					
C						1	1

and taking the transpose of the square section and ORing it with the original square section results in:

	A	B	E	A	C
A		1	1		1
B	1		1	1	
E	1	1		1	
A		1	1		
C	1				

Now removing all ones except those most closely surrounding the diagonal one gets:

	A	B	E	A	C
A		1			
B	1		1		
E		1		1	
A			1		
C	1				

The result is that in each row the ones enclose the range of each statement. In this example the 5-th statement can be interchanged with either the 2-nd, 3-rd, or 4-th statements. The further questions concerning consideration of the total effect of a number of order changes and the incorporation of indices are opened by Kuck [18].

C.2 Translation Decisions

In TRANQUIL Parallelism can be explicitly stated using the SIM control statement, where each statement S_i in its scope is executed using the data available before any part of is executed. In examining, in particular, a FORTRAN DO loop a decision must be made as to whether a SIM control statement can be used to replace the DO, this task requiring knowledge about essential ordering. Consider the example of matrix multiplication:

```
DO 20 I = 1, 200
DO 20 J = 1,  20
C[I, J] = 0
DO 20 K = 1, 300
20 C[I, J]: = C[I, J] + A[I, K] * B[K, J]
```

The possibilities that exist without considering the arithmetic are 8 in number (SIM or SEQ for each DO). How, then, does one rule out SIM execution? The first criterion is whether a statement in the scope of the loop can be done in parallel as determined from statement and context analysis. This is the case for both statements of the routine if I and J are run simultaneously and K

sequentially. The next criterion is relating the storage structure of the arrays involved. Again in the example we are content if matrices A, B and C are stored straight or skewed. A last criterion for the present is the size of the index sets associated with the control variables. In the example, since I is associated with a set of 200 elements and J with a set of only 20 elements one chooses to run I simultaneously. Thus we have the following TRANQUIL program segment:

```
II := [1, 2,..., 200];
JJ := [1, 2,..., 20];
KK := [1, 2,..., 300];
FOR (I) SIM (II) DO
    FOR (J) SEQ (JJ) DO
    BEGIN
    C[I, J] ← 0;
    FOR (K) SEQ (KK) DO
    C[I, J] ← C[I, J] + A[I, J] × B[K, J];
    END
```

It should be noted that mere replacement of sequential loops by parallel ones is not enough. In the following program I can be run simultaneously and J sequentially for the 1-st statement and J can be run simultaneously only for the 2-nd.

```
DO 15 I = 1, 200
DO 15 J = 20, 60
A[I, J] := B[I, J] + A[I, J-1]
15 C[J] := C[J-1]
```

It should also be noted that the last statement can be done in parallel because of the routing capabilities of ILLIAC IV.

Another more complicated program may further illustrate what to watch for in translation. The table look-up problem involves a table like the one below with value ordered temperatures and pressures as the co-ordinates.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7
T_5	1	5				21	
T_4	2	6				22	
T_3	3					23	
T_2	4					24	
T_1							

The task is to find out what area an experimental points falls in. A procedure to solve this simply involves noting when a test value TT becomes less than a table value where the table values are checked in ascending order (see [19]). A FORTRAN IV program to do part of this for 256 points is

```
DO 30 J = 1, 256
DO 20 I = 2, 5
IF (TT[J] < T[I])  GO TO 25
20 CONTINUE
25 A[J] := (I-1) * 4
30 CONTINUE
```

Note the exit from the inner loop on test completion. The first questions are can TT[J] < T[I] be done simultaneously for I and J in the context of the loop it is in and can A[J] := (I-1) * 4 be done simultaneously. In the latter case I is not a subscript and thus I is out of the running for this statement. Now we note that the transfer out of the inner loop can be taken care of by mode setting via index sets on ILLIAC IV and that since I is so small we do not run it simultaneously for the 1st statement either. Thus J can run simultaneously over both statements if A and TT are stored across ILLIAC IV memory. We can develop the following TRANQUIL program segment:

```
II ← [2, 3,..., 5]
JJ ← [1, 2,..., 256]
FOR (I) SEQ (II) DO
    BEGIN
    FOR (J) SIM (JJ) DO
        BEGIN
        SS ← SET[J: TT[J] < T[I]];
        FOR (S) SIM (SS) DO
            A[S] ← I;
        IFFOR ANY TT[J] > T[I] THEN
            SETEMP ← SET[J: TT[J] > T[I]]
            ELSE
            GO TO TTEND;
        END;
    JJ ← SETEMP;
    END;
TTEND: A[JJ] ← (A[JJ] - 1) × 4
```

It is clear that the problem of translating a serial language like FORTRAN into a parallel language like TRANQUIL involves close inspection of statement relationships, control, storage schemes, and machine capabilities as in the examples above.

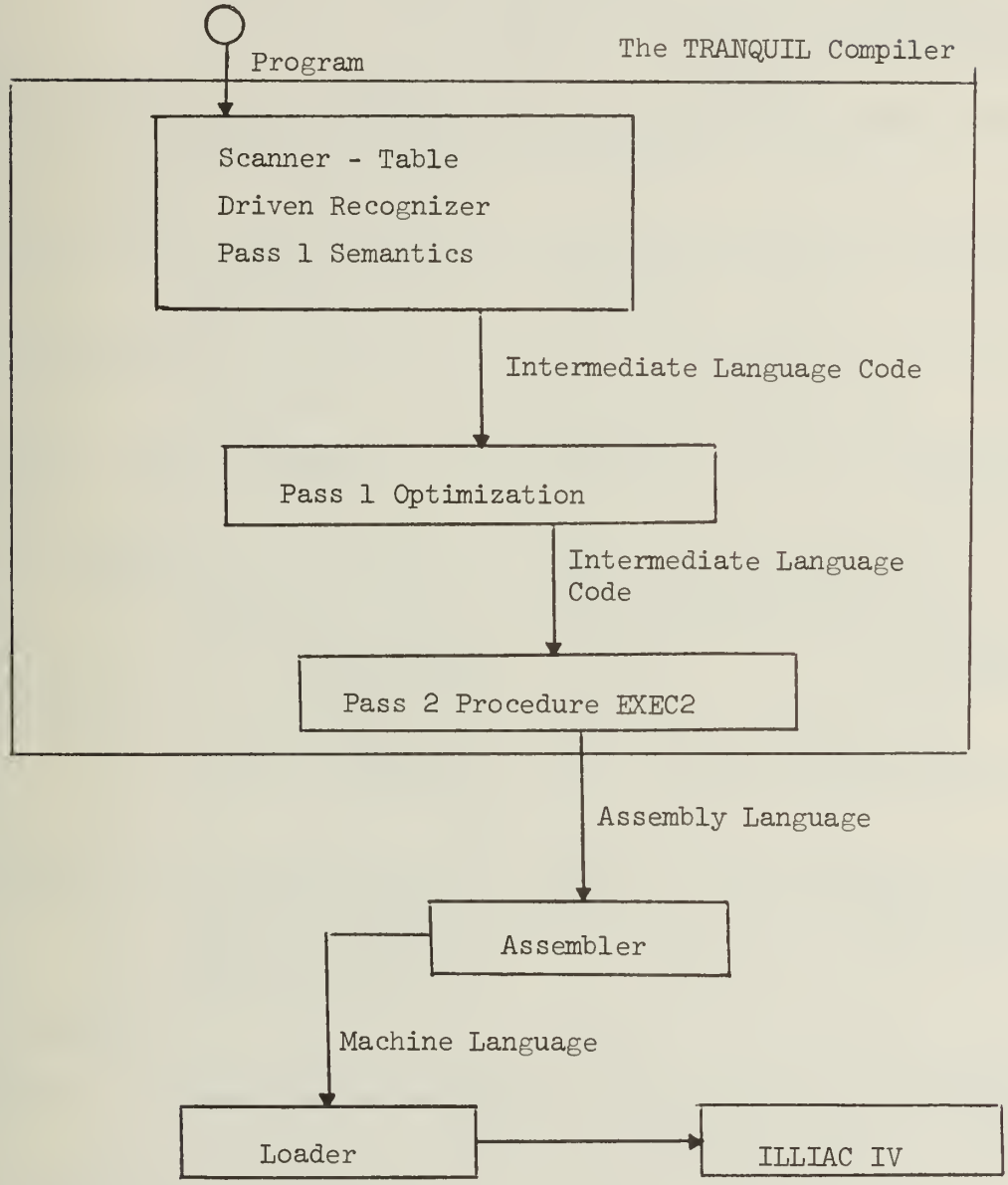
APPENDIX D

A SAMPLE TRANQUIL PROGRAM

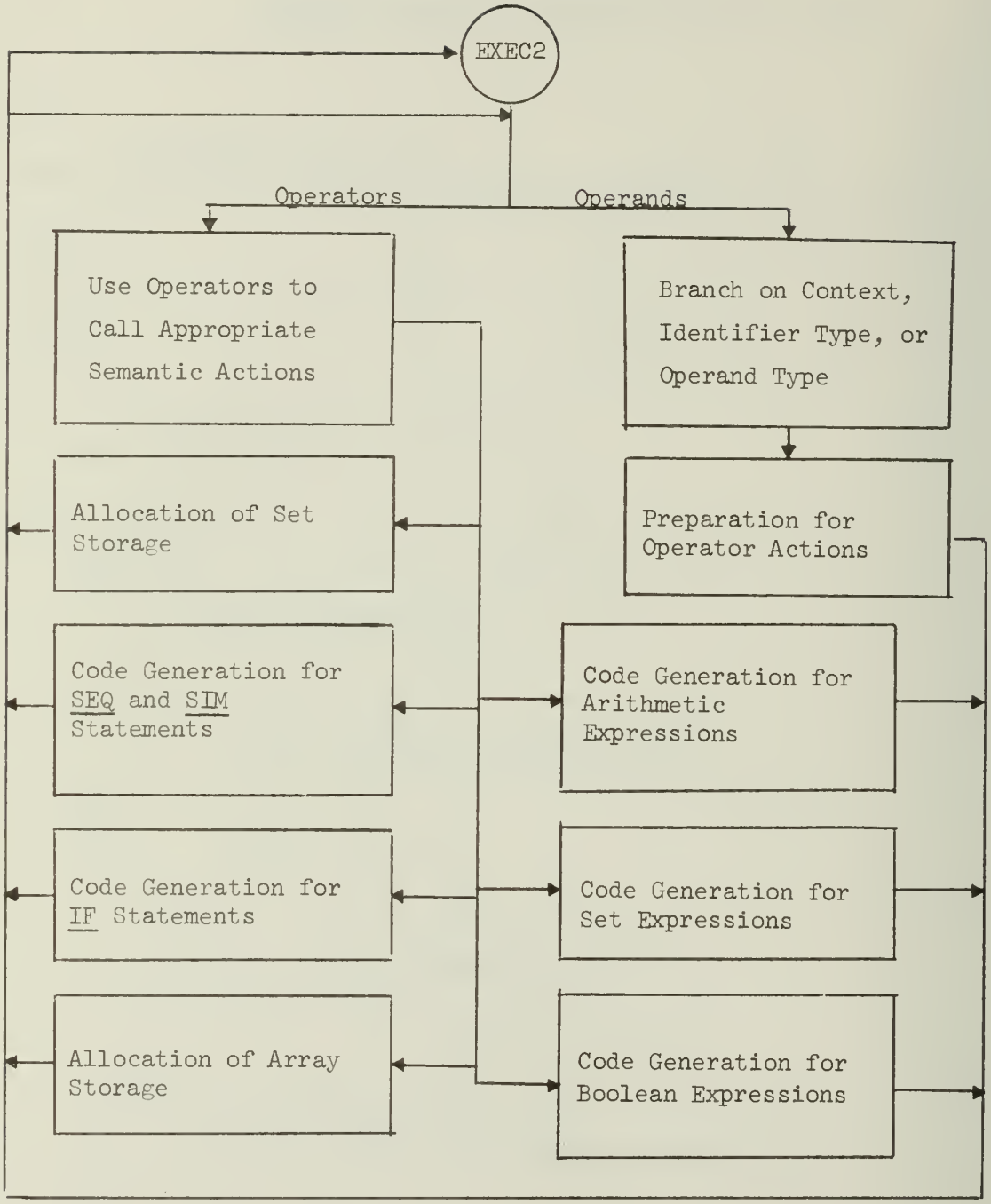
```
BEGIN
REAL SKEWED ARRAY A, B[0:100,0:100];
INCSET JJ;
MONOSET II(1)[27:6], KK(1)[100:60];
INTEGER  I,J,K,L;
II ← [2, 10, 13, 21, 24];
JJ ← [2, 4,..., 98];
FOR (I) SEQ (II) DO
     BEGIN FOR (J) SIM (JJ) DO
               KK ← SET (J:A[I, J] < B[I, J]) ;
          FOR (K) SIM (KK) DO
               A[I, K] ← A[I+1, K] + B[I, K+1]
     END;
     BEGIN;
          MONOSET LL [50:50];
          REAL STRAIGHT ARRAY C[0:100,0:100];
          LL ← [1, 2,..., 49]
          FOR (L, J) SIM (LL,JJ) DO
               C[J, L] ← C[J, L+1] + C[J+1, L]
     END;
     FOR (I, K) SIM (II × KK) DO
          A[I, K] ← A[I+1, K] + B[I, K]
     END
END
```

APPENDIX E

GENERAL FLOW CHARTS FOR THE TRANQUIL COMPILER



From a Program to ILLIAC IV
via the TRANQUIL Compiler



General Simplified Structure of EXEC2

LIST OF REFERENCES

[1] Gosden, J. A., "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni- Processor Computers," 1966 Fall Joint Comp. Conf., AFIPS Proc., vol. 29. Washington, D.C.: Spartan, 1966, pp. 651-660.

[2] Bernstein, A. J., "Analysis of Programs for Parallel Processing," IEEE Trans. Electronic Computers, vol. EC-15, pp. 757-763, October 1966.

[3] Opler, A., "Procedure-Oriented Language Statements to Facilitate Parallel Processing," Comm. ACM, vol. 8, pp. 306-307, May 1965.

[4] Anderson, J. P., "Program Structures for Parallel Processing," Comm. ACM, vol. 8, pp. 786-788, December 1965.

[5] Wirth, N., A Note on "Program Structures for Parallel Processing" by Anderson, Comm. ACM, vol. 9, pp. 320-321, May 1966.

[6] Constantine, L. L., "Control of Sequence and Parallelism in Modular Programs," 1968 Spring Joint Comp. Conf., AFIPS Proc., vol. 32. Washington, D. C.,: Thompson Book Company, 1968, pp. 409-414.

[7] Karp, R. M. and Miller, R. C., "Properties of a Model for Parallel Computations: Determinacy, Termination, Queuing," IBM Watson Research Center, Yorktown Heights, N. Y., RC-1285, September 1964.

[8] Martin, D. C. and Estrin, G., "Models of Computational Systems - Cyclic to Acyclic Graph Transformations," IEEE Trans. Electronic Computers, vol. EC-16, pp. 70-79, February 1967.

[9] Bingham, H. W., Fisher, D. A., and Simon, W. L., "Detection of Implicit Computational Parallelism from Input-Output Sets," Technical Report ECOM-02463-1, Burroughs TR-66-4, AD645438, Paoli, Pennsylvania, December 1966.

[10] Bingham, H. W. and Reigel, E. W., "Parallelism in Computer Programs and in Machines," Technical Report ECOM-02463-6, Burroughs TR-68-1, Paoli, Pennsylvania, April 1968.

[11] Bingham, H. W., Reigel, E. W., and Fisher, D. A., "Parallelism in Computer Programs and Multiprocessing Machine Organizations," Technical Report ECOM-02463-5, Burroughs TR-67-6, Paoli, Pennsylvania, January 1968.

[12] Kuck, D. J., "ILLIAC IV Software and Application Programming," IEEE Trans. on Computers, vol. C-17, pp. 758-770, August 1968.

[13] Budnik, P., "TRANQUIL Arithmetic," M.S. Thesis, Dept. of Computer Science, University of Illinois, January 1969.

[14] Muraoka, Y., "Storage Allocation Algorithms in the TRANQUIL Compiler," M.S. Thesis, Department of Computer Science, University of Illinois, January 1969.

[15] Naur, P. (Ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, vol. 6, pp. 1-17, January 1963.

[16] Wells, M. B., "Aspects of Language Design for Combinatorial Computing," IEEE Trans. Electronic Computers, vol. EC-13, pp. 431-438, August 1964.

[17] Iverson, K. E., A Programming Language. New York: Wiley, 1962.

[18] Kuck, D. J., "Trip Report," Department of Computer Science, University of Illinois, August 1, 1968, ILLIAC IV (unpublished).

[19] Wilhelmson, R., "Table Lookup Problem and Solutions," Department of Computer Science, University of Illinois, March 1967, ILLIAC IV Doc. 113 (unpublished).

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Department of Computer Science University of Illinois Urbana, Illinois 61801	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP

3. REPORT TITLE

CONTROL STATEMENT SYNTAX AND SEMANTICS OF A LANGUAGE FOR PARALLEL PROCESSORS

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Research Report

5. AUTHOR(S) (First name, middle initial, last name)

Robert B. Wilhelmson

6. REPORT DATE January 13, 1969	7a. TOTAL NO. OF PAGES 86	7b. NO. OF REFS 19
8a. CONTRACT OR GRANT NO. 46-26-15-305 b. PROJECT NO. USAF 30(602)4144 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) DCS Report No. 298 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DCS.

11. SUPPLEMENTARY NOTES NONE	12. SPONSORING MILITARY ACTIVITY Rome Air Development Center Griffiss Air Force Base Rome, New York 13440

13. ABSTRACT

A description of the control statement features of TRANQUIL, a language for describing algorithms in terms of parallel constructs, and their implementation in a compiler for the parallel array computer ILLIAC IV is given. This includes descriptions of the revised ALGOL loop specification, simultaneous execution specification, and set specification; and the implementation of the storage and use of sets as well as use of the control unit registers. The concept of parallelism in a language is discussed and the problems of automatic translation of a serial to a parallel language brought out.

D FORM 1 NOV 68 1473

UNCLASSIFIED

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT.	ROLE	WT	ROLE	WT
Sequential Control: The SEQ Statement						
Simultaneous Control: The SIM Function and the SIM Statement						
Nested SEQ and SIM Statements						
Compiler Implementation of Control Statements						
Metalanguage for Specifying Syntax						
Tranquil Set and Control Statement Syntax						